UNIT 1

1. What is network protocol? With a neat block diagram explain the network application for client and server.
2. List out the various approaches used to handle multiple clients at the same time.
3. With a neat block diagram, explain the client and server communication on Local Area Network using TCP.
4. With a neat block diagram, explain the client and server communication over Wide Area Network using TCP.
5. List and explain the steps involved in simple daytime client.
6. Develop the 'C' program to implement simple daytime client.
7. Comment on the Protocol Independence. Modify the day time client program for IPv6.
8. What are wrapper functions? Develop the wrapper function for the following:
    a. Socket function
    b. Pthread_mutex_lock
9. List and explain the steps involved in simple daytime server.
10. Develop the 'C' program to implement simple daytime server.
11. Write a note on Unix errno value.
12. Explain with a neat block diagram the layers of OSI model and Internet protocol suite.
13. Write a note on 64- bit architectures.
14. Explain the features of the following protocols:
    a. IPv4
    b. IPv6
    c. TCP
    d. UDP
    e. SCTP
    f. ICMP
    g. IGMP
    h. ARP
    i. RARP
    j. ICMPv6
    k. BPF
    l. DLPI
15. List and explain the features of UDP Protocol in detail.
16. List and explain the features of TCP Protocol in detail.
17. Explain with a neat diagrams the following:
    a. TCP connection establishment
    b. TCP data transfer
    c. TCP connection termination
18. Explain the TCP State Transition diagram with a neat diagram.

Unit 2: (Sockets Introduction)

1. With a standard POSIX definition explain socket address porotype for IPv4 *(sockaddr_in)* and IPv6 *(sockaddr_in6)*
2. With a standard POSIX definition explain socket address structure porotype: *sockaddr*
3. Show the prototype for storage socket address structure: *sockaddr_storage*
4. Compare the various socket address structures: *sockaddr_in (), sockaddr_in6.*
5. What are Value- Result Arguments? Explain the scenario with a neat block diagram.
6. Explain the functions which passes socket address structure from the process to the kernel with a neat block diagram.
7. Explain with a neat diagram the various byte ordering functions.
8. Write a note on the Byte Manipulation Functions.
9. Develop a 'C' program to determine host byte order.
10. Illustrate the significance of socket functions for elementary TCP client/server with a neat block diagram.
11. Explain the following arguments of the socket function:
    a. Family
    b. Type
    c. Protocol
12. Explain the following functions of TCP socket:
    a. connect
    b. bind
    c. listen
    d. accept
    e. close
13. With a neat block diagram explain the queues maintained by TCP for a listening socket. Also show the packets exchanged during the connection establishment with these two queues.
14. Illustrate the significance of fork and exec functions.
15. Outline the typical concurrent server with the help of pseudocode.
16. Demonstrate the status of client/ server before and after call to *accept* returns with a neat block diagram.
17. Demonstrate the status of client/ server after fork returns with a neat block diagram.
18. Demonstrate the status of client/ server after parent and child close appropriate sockets with a neat block diagram.
19. Comment on the significance of *getsockname* and *getpeername* functions.
20. Develop the pseudocode that returns the address family of a socket.

Self-Learning Topics:

21. List and explain with a neat block diagram the steps associated with simple TCP echo client and server.
22. Develop the 'C' program to demonstrate the TCP echo server: main function
23. Develop the 'C' program to demonstrate the TCP echo server: str_echo function
24. Develop the 'C' program to demonstrate the TCP echo client: main function
25. Develop the 'C' program to demonstrate the TCP echo client: str_cli function

Unit 3: (Elementary UDP Sockets)

1. Illustrate the significance of socket functions for UDP TCP client/server with a neat block diagram.
2. Explain the following functions of UDP socket:
    a. recvfrom
    b. sendto
3. List and explain with a neat block diagram the steps associated with simple UDP echo client and server.
4. Develop the 'C' program to demonstrate the UDP echo server: main function
5. Develop the 'C' program to demonstrate the UDP echo server: dg_echo function
6. Develop the 'C' program to demonstrate the UDP echo client: main function
7. Develop the 'C' program to demonstrate the UDP echo client: dg_cli function
8. Outline the summary of TCP client/server with two clients.
9. Outline the summary of UDP client/server with two clients.
10. Develop the 'C' program for dg_cli function that verifies returned socket address.
11. Outline the summary of UDP client/server from client's perspective with a neat block diagram.
12. Outline the summary of UDP client/server from server's perspective with a neat block diagram.
13. Develop the 'C' program to demonstrate the UDP dg_cli function that calls connect.
14. Develop the 'C' program to demonstrate the UDP dg_cli function that writes a fixed number of datagrams to the server.
15. Develop the 'C' program to demonstrate the UDP dg_echo function that counts received datagrams.
16. Develop the 'C' program to demonstrate the UDP dg_echo function that increases the size of the socket receive queue.
17. Develop the 'C' program for UDP that uses connect to determine outgoing interface.
18. Make use of select function for TCP and UDP Echo server.
19. Illustrate the significance of socket functions for SCTP using one-to-one style with a neat block diagram.
20. Illustrate the significance of socket functions for SCTP using one-to-many style with a neat block diagram.
21. With a neat block diagram explain the shutdown function to close an SCTP association.

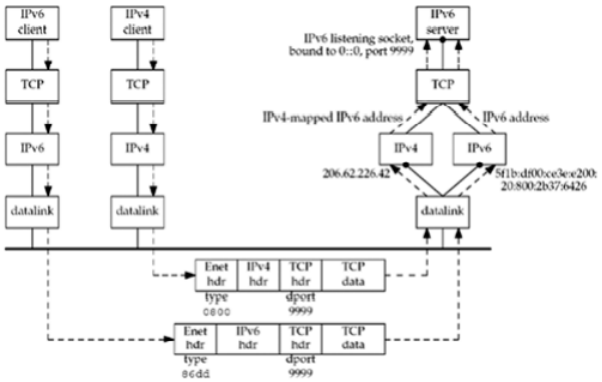
Self-Learning Topics:

22. Explain the simple SCTP streaming echo client and server with a neat block diagram.

Unit 4: (Advanced Sockets-I)

1. With a neat block diagram explain IPv6 server on dual stack host serving IPv4 and IPv6 clients.

Figure 12.2. IPv6 server on dual-stack host serving IPv4 and IPv6 clients.



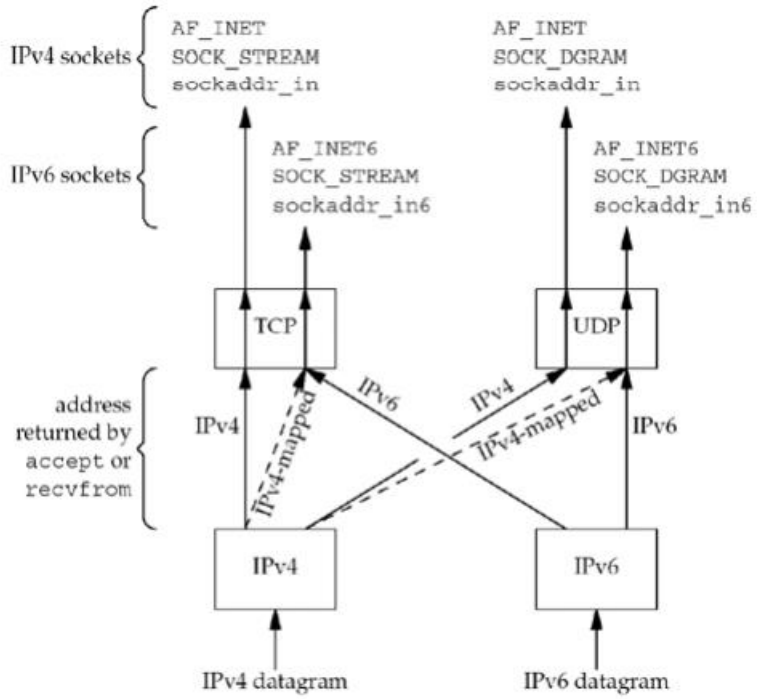
2. We have an IPv4 client and an IPv6 client on the left. The server on the right is written using IPv6 and it is running on a dual-stack host.
3. The server has created an IPv6 listening TCP socket that is bound to the IPv6 wildcard address and TCP port 9999.
4. We assume the clients and server are on the same Ethernet. They could also be connected by routers, as long as all the routers support IPv4 and IPv6, but that adds nothing to this discussion
5. We can summarize the steps that allow an IPv4 TCP client to communicate with an IPv6 server as follows

   - The IPv6 server starts, creates an IPv6 listening socket, and we assume it binds the wildcard address to the socket.
   - The IPv4 client calls gethostbyname and finds an A record for the server. The server host will have both an A record and a AAAA record since it supports both protocols, but the IPv4 client asks for only an A record.
   - The client calls connect and the client's host sends an IPv4 SYN to the server.
   - The server host receives the IPv4 SYN directed to the IPv6 listening socket, sets a flag indicating that this connection is using IPv4-mapped IPv6 addresses, and responds with an IPv4 SYN/ACK. When the connection is established, the address returned to the server by accept is the IPv4-mapped IPv6 address.
   - When the server host sends to the IPv4-mapped IPv6 address, its IP stack generates IPv4 datagrams to the IPv4 address. Therefore, all communication between this client and server takes place using IPv4 datagrams.
   - . Unless the server explicitly checks whether this IPv6 address is an IPv4-mapped IPv6 address (using the IN6_IS_ADDR_V4MAPPED macro described in Section 12.4), the server never knows that it is communicating with an IPv4 client. The dual-protocol stack handles this detail. Similarly, the IPv4 client has no idea that it is communicating with an IPv6 server

The scenario is similar for an IPv6 UDP server, but the address format can change for each datagram. For example, if the IPv6 server receives a datagram from an IPv4 client, the address returned by

recvfrom will be the client's IPv4-mapped IPv6 address. The server responds to this client's request by calling sendto with the IPv4-mapped IPv6 address as the destination. This address format tells the kernel to send an IPv4 datagram to the client. But the next datagram received for the server could be an IPv6 datagram, and recvfrom will return the IPv6 address. If the server responds, the kernel will generate an IPv6 datagram

2) Explain with a neat block diagram how the received IPv4 and IPv6 datagrams are processed depending on the type of receiving socket.

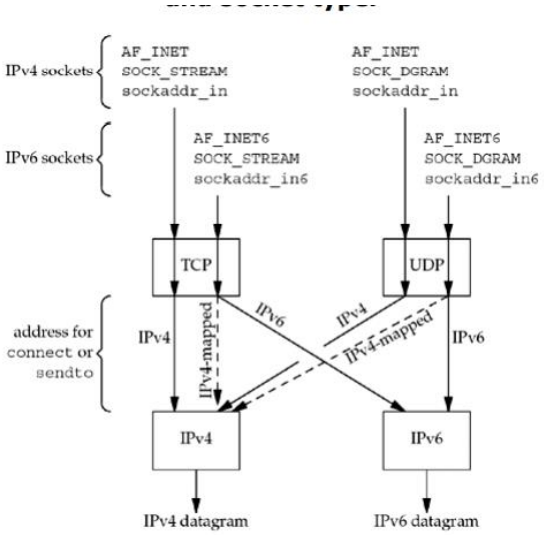


- ✓ If an IPv4 datagram is received for an IPv4 socket, nothing special is done. These are the two arrows labeled "IPv4" in the figure: one to TCP and one to UDP. IPv4 datagrams are exchanged between the client and server.
- ✓ If an IPv6 datagram is received for an IPv6 socket, nothing special is done. These are the two arrows labeled "IPv6" in the figure: one to TCP and one to UDP. IPv6 datagrams are exchanged between the client and server.
- ✓ When an IPv4 datagram is received for an IPv6 socket, the kernel returns the corresponding IPv4-mapped IPv6 address as the address returned by accept (TCP) or recvfrom (UDP). These are the two dashed arrows in the figure. This mapping is possible because an IPv4 address can always be represented as an IPv6 address. IPv4 datagrams are exchanged between the client and server.
- ✓ The converse of the previous bullet is false: In general, an IPv6 address cannot be represented as an IPv4 address; therefore, there are no arrows from the IPv6 protocol box to the two IPv4 sockets

Most dual-stack hosts should use the following rules in dealing with listening sockets:

- A listening IPv4 socket can accept incoming connections from only IPv4 clients.
- If a server has a listening IPv6 socket that has bound the wildcard address and the IPV6_V6ONLY socket option (Section 7.8) is not set, that socket can accept incoming connections from either IPv4 clients or IPv6 clients. For a connection from an IPv4 client, the server's local address for the connection will be the corresponding IPv4-mapped IPv6 address.
- If a server has a listening IPv6 socket that has bound an IPv6 address other than an IPv4-mapped IPv6 address, or has bound the wildcard address but has set the IPv6_V6ONLY socket option (Section 7.8), that socket can accept incoming connections from IPv6 clients only

3) Explain with a neat block diagram how the client requests are processed depending on the address type and socket type.



- If an IPv4 TCP client calls connect specifying an IPv4 address, or if an IPv4 UDP client calls sendto specifying an IPv4 address, nothing special is done. These are the two arrows labeled "IPv4" in the figure
- .If an IPv6 TCP client calls connect specifying an IPv6 address, or if an IPv6 UDP client calls sendto specifying an IPv6 address, nothing special is done. These are the two arrows labeled "IPv6" in the figure.
- If an IPv6 TCP client specifies an IPv4-mapped IPv6 address to connect or if an IPv6 UDP client specifies an IPv4-mapped IPv6 address to sendto, the kernel detects the mapped address and causes an IPv4 datagram to be sent instead of an IPv6 datagram. These are the two dashed arrows in the figure.
- An IPv4 client cannot specify an IPv6 address to either connect or sendto because a 16-byte IPv6 address does not fit in the 4-byte in_addr structure within the IPv4 sockaddr_in structure. Therefore, there are no arrows from the IPv4 sockets to the IPv6 protocol box in the figure

the conversion of the IPv4 address to the IPv4-mapped IPv6 address is done by the resolver according to the rules in Figure 11.8, and the mapped address is then passed transparently by the application to connect or sendto.

4) List and explain the numerous ways to start a daemon.

There are numerous ways to start a daemon:

- During system startup, many daemons are started by the system initialization scripts. These scripts are often in the directory /etc or in a directory whose name begins with /etc/rc, but their location and contents are implementation-dependent. Daemons started by these scripts begin with superuser privileges.
  A few network servers are often started from these scripts: the inetd superserver (covered later in this chapter), a Web server, and a mail server (often sendmail). The syslogd daemon that we will describe in Section 13.2 is normally started by one of these scripts.
- Many network servers are started by the inetd superserver. inetd itself is started from one of the scripts in Step 1. inetd listens for network requests (Telnet, FTP, etc.), and when a request arrives, it invokes the actual server (Telnet server, FTP server, etc.).
- The execution of programs on a regular basis is performed by the cron daemon, and programs that it invokes run as daemons. The cron daemon itself is started in Step 1 during system startup.
- The execution of a program at one time in the future is specified by the at command. The cron daemon normally initiates these programs when their time arrives, so these programs run as daemons.
- Daemons can be started from user terminals, either in the foreground or in the background. This is often done when testing a daemon, or restarting a daemon that was terminated for some reason

5) List and explain the actions on startup for syslogd Daemon.

Unix systems normally start a daemon named syslogd from one of the system initializations scripts, and it runs as long as the system is up. Berkeley-derived implementations of syslogd perform the following actions on startup:

1. The configuration file, normally /etc/syslog.conf, is read, specifying what to do with each type of log message that the daemon can receive. These messages can be appended to a file (a special case of which is the file /dev/console, which writes the message to the console), written to a specific user (if that user is logged in), or forwarded to the syslogd daemon on another host.

2. A Unix domain socket is created and bound to the pathname /var/run/log (/dev/log on some systems).

3. A UDP socket is created and bound to port 514 (the syslog service).

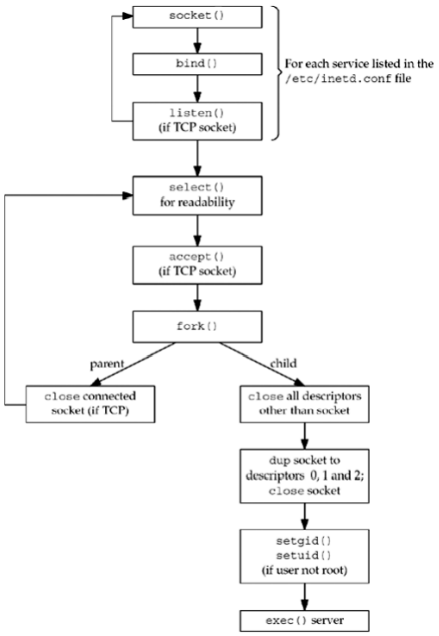
4. The pathname /dev/klog is opened. Any error messages from within the kernel appear as input on this device

6. With a function prototype explain the syslog Function.

Self- Learning Topics:

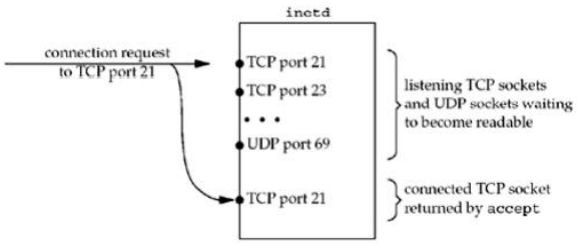
7. Explain the significance of daemon_init function.
8. List and explain the steps performed by of inetd Daemon.

Figure 13.7. Steps performed by `inetd`.



- On startup, it reads the /etc/inetd.conf file and creates a socket of the appropriate type (stream or datagram) for all the services specified in the file. The maximum number of servers that inetd can handle depends on the maximum number of descriptors that inetd can create. Each new socket is added to a descriptor set that will be used in a call to select
- . 2. bind is called for the socket, specifying the port for the server and the wildcard IP address. This TCP or UDP port number is obtained by calling getservbyname with the service-name and protocol fields from the configuration file as arguments.
- For TCP sockets, listen is called so that incoming connection requests are accepted. This step is not done for datagram sockets.
- After all the sockets are created, select is called to wait for any of the sockets to become readable. Recall from Section 6.3 that a listening TCP socket becomes readable when a new connection is ready to be accepted and a UDP socket becomes readable when a datagram arrives. inetd spends most of its time blocked in this call to select, waiting for a socket to be readable.
- When select returns that a socket is readable, if the socket is a TCP socket and the nowait flag is given, accept is called to accept the new connection.
- The inetd daemon forks and the child process handles the service request. This is similar to a standard concurrent server (Section 4.8)
  - If the socket is a stream socket, the parent process must close the connected socket (like our standard concurrent server). The parent calls select again, waiting for the next socket to become readable.

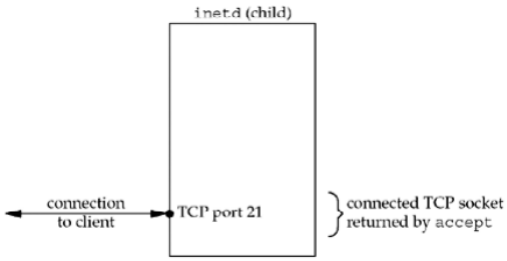
9. With a neat block diagram explain the inetd descriptors when connection request arrives for TCP port.



The connection request is directed to TCP port 21, but a new connected socket is created by accept.

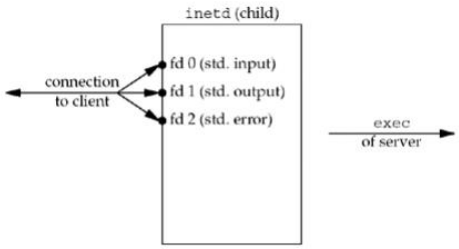
Figure 13.9 shows the descriptors in the child, after the call to fork, after the child has closed all the descriptors except the connected socket

Figure 13.9. `inetd` descriptors in child.



The next step is for the child to duplicate the connected socket to descriptors 0, 1, and 2 and then close the connected socket. This gives us the descriptors shown in Figure 13.10

Figure 13.10. `inetd` descriptors after `dup2`.



The child then calls exec. Recall from Section 4.7 that all descriptors normally remain open across an exec, so the real server that is execed uses any of the descriptors, 0, 1, or 2, to communicate with the client. These should be the only descriptors open in the server

10. Explain the following with a neat block diagram:
    a. inetd descriptors in child
    b. inetd descriptors after dup2

11. Develop the pseudocode for daemon_inetd function to daemonize process run by inetd.

Figure 13.11 `daemon_inetd` function: daemonizes process run by `inetd`.

daemon_inetd.c

```
1 #include     "unp.h"
2 #include     <syslog.h>

3 extern int daemon_proc;         /* defined in error.c */

4 void
5 daemon_inetd(const char *pname, int facility)
6 {
7     daemon_proc = 1;            /* for our err_XXX() functions */
8     openlog(pname, LOG_PID, facility);
9 }
```

Figure 13.12 Protocol-independent daytime server that can be invoked by `inetd`.

inetd/daytimetcpsrv3.c

```
 1 #include     "unp.h"
 2 #include     <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     socklen_t len;
 7     struct sockaddr *cliaddr;
 8     char    buff[MAXLINE];
 9     time_t  ticks;

10     daemon_inetd(argv[0], 0);

11     cliaddr = Malloc(sizeof(struct sockaddr_storage));
12     len = sizeof(struct sockaddr_storage);
13     Getpeername(0, cliaddr, &len);
14     err_msg("connection from %s", Sock_ntop(cliaddr, len));

15     ticks = time(NULL);
16     snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));


17      Write(0, buff, strlen(buff));

18      Close(0);                   /* close TCP connection */
19      exit(0);
20 }
```

Unit 5: (Advanced Sockets-II)

1. Outline the important points with respect to unicasting, multicasting and broadcasting.
2. List and explain the routing protocols which makes use of multicasting.
3. Differentiate between unicast and broadcast.
4. Make use of UDP datagram to understand unicasting.
5. Make use of UDP datagram to understand broadcasting.
6. Explain with a neat block diagram how IPv4 and IPv6 multicast addresses are mapped to Ethernet addresses.
7. What is the format of IPv6 multicast addresses?
8. Illustrate the scope of multicast addresses.
9. Illustrate with a neat block diagram, multicast example of a UDP datagram.
10. Illustrate the NTP packet format and definitions with the help of ntp.h header.